

# Prolog

Mattox Beckman  
beckman@iit.edu

Illinois Institute of Technology

Question: How do you decide truth?

- Start with some *objects*  
“socrates”, “john”, “mary”
- Write down some *facts* (true statements) about those objects.
  - Facts express either properties of the object, or  
“socrates is human”
  - relationship to other objects.  
“mary likes john”
- Write down some *rules* (facts that are true if other facts are true).  
“if X is human then X is mortal”
- Facts and Rules can become *predicates*.  
“is socrates mortal?”

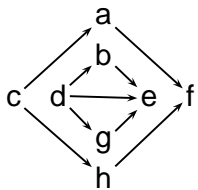
In this lecture we will introduce Prolog.

- Be able to explain the models of data and program for Prolog.  
(The Two Questions)
- Be able to write some simple programs in Prolog.
- Know how to use Prolog’s arithmetic operations.
- Know how to use lists and patterns.

First Order Predicate Logic is one system for encoding these kinds of questions.

- Predicate means that we have functions that take objects and return “true” or “false”.  
*human(socrates)*.
- Logic means that we have *connectives* like and, or, not, and implication.
- First Order means that we have variables (created by “for all” and “there exists”), but that they only work on objects.  
 $\forall X . \text{human}(X) \rightarrow \text{mortal}(X)$ .

- Starting point: First Order Predicate Logic.
- Realization: Computers can reason with this kind of logic.
- Impetus was the study of *mechanical theorem proving*
- Developed in 1970 by Alain Colmerauer and Rober Kowalski and others.
- Uses: databases, expert systems, AI.



```

1 human(socrates).
2 fatherof(socrates,jane).
3 fatherof(zeus,apollo).
  
```

```

1 connected(c,a).
2 connected(a,f).
3 connected(d,b).
4 connected(b,e).
5 connected(d,e).
6 connected(d,g).
7 connected(g,e).
8 connected(c,h).
9 connected(h,f).
  
```

What is the nature of data?

Prolog data consists of *facts* about *objects* and logical *rules*.

What is the nature of a program?

A program in Prolog is a set of facts and rules, followed by a *query*.

```

1 mortal(X) :- human(X).
2 human(Y) :- fatherof(X,Y), human(X).
3
4 pathfrom(X,Y) :- connected(X,Y).
5 pathfrom(X,Y) :- connected(X,Z), pathfrom(Z,Y).
  
```

- Capital letters are variables.
  - Appearing left of :- means “for all”
  - Appearing right of :- means “there exists”

$$\forall x. human(x) \rightarrow mortal(x).$$

$$\forall y. (\exists x. fatherof(x,y) \wedge human(x)) \rightarrow human(y)$$

Programs are executed by searching the database and attempting to perform unification.

```
?- human(socrates).    listed, therefore true
?- mortal(socrates).   not listed
```

Relevant rules:

```
human(socrates).
human(Y) :- fatherof(X,Y), human(X).
mortal(X) :- human(X).
```

Socrates is not listed as being mortal, but `mortal(socrates)` unifies with `mortal(X)` if we replace `X` with `socrates`. This gives us a *subgoal*. Replace `X` with `socrates` and try it....

```
?- mortal(jane). not in database
but we have mortal(X) :- human(X).
so we substitute mortal(jane) :- human(jane).
subgoal: human(jane). not there either
but we have human(Y) :- fatherof(X,Y), human(X).
so we substitute
    human(jane) :- fatherof(X,jane), human(X).
subsubgoal1 fatherof(X,jane).
we find fatherof(socrates,jane)
so try the next subgoal
subsubgoal2 human(socrates). yes
therefore human(jane). is true
therefore mortal(jane). is true
```

Replace `X` with `socrates` in this rule:

```
mortal(X) :- human(X).
```

to get

```
mortal(socrates) :- human(socrates).
```

Since `human(socrates)` is in the database, we know that `mortal(socrates)` is also true.

- Prolog can also give you a list of elements that make a predicate true. Remember unification.

```
1 ?- fatherof(Who,apollo).
2 Who = zeus ;
3
4 ?- pathfrom(c,X).
5 X = a ;
6 X = h ;
7 X = f ;
8 X = f ;
9 No
```

The semicolon is entered by the user— it means to keep searching.

```
?- pathfrom(c,X).
→ pathfrom(c,Y) :- connected(c,Y).
X = a ;
```

When we hit semicolon, we tell it to pretend that this is false, and to keep searching. We have to throw away the `connected(c,a)` result, so we *backtrack* through our database to try again.

```
→ pathfrom(c,Y) :- connected(c,Y).
X = h ;
```

We tell it to try again with this one, too. At this point, we no longer have any rules that say that `c` is connected to something.

```
1 fact(0,1).
2 fact(N,X) :- M is N-1, fact(M,Y), X is Y * N.
3 ?- fact(5,X).
```

- Unify `fact(5,X)` with `fact(N,X)`.  
`fact(5,X) :- M is 5-1, fact(M,Y), X is Y * 5.`
- Next compute `M`.  
`fact(5,X) :- 4 is 5-1, fact(4,Y), X is Y * 5.`
- Recursive call sets `Y` to 24.  
`fact(5,X) :- 4 is 5-1, fact(4,24), X is 24 * 5.`
- Compute `x`  
`fact(5,120) :- 4 is 5-1, fact(4,24), 120 is 24 * 5.`

```
pathfrom(c,Y) :- connected(c,Z), pathfrom(Z,Y).
```

We will first find something in the database that says that `c` is connected to some `z`, and then check if there is a path between `z` and `Y`.

We find `a` and `h` as last time. When we check `a`, we check for `pathfrom(a,Y)`, and find that `connected(a,f)` is in the database. The same thing happens for `h`, which is why `f` is reported as an answer twice.

Order of sub-goals is important! Why does this happen?

```
1 badfact(0,1).
2 badfact(N,X) :- badfact(M,Y), M is N-1, X is Y * N.
3 ?- badfact(5,X).
4 ERROR: Arguments are not sufficiently instantiated
5 ^ Exception: (8) 0 is _G278-1 ?
```

```
badfact(5,X) :- badfact(M,Y), M is 5-1, X is Y * 5.
badfact(5,X) :- badfact(0,1), 0 is 5-1, X is 1 * 5.
                        (oops)
```

Prolog lists are very similar to CaML lists.

- Empty list: `[]`
- Singleton list: `[x]`
- List with multiple elements: `[x,y,[a,b],c]`
- Head and tail representation `[H|T]`

Differences:

- Prolog lists are *not* monotonic!

```
1 sumlist([],0).
2 sumlist([H|T],X) :- sumlist(T,Y), X is Y + H.
3
4 ?- sumlist([2,3,4,5],X).
5 X = 14
```

The `length` predicate is built in.

```
1 mylength([],0).
2 mylength([H|T],X) :- mylength(T,Y), X is Y + 1.
3
4 ?- mylength([2,3,4,5],X).
5 X = 4 ;
6 No
```

This example looks like `badfact`, in that the `is` clause happens after the recursion. Why is this safe?

```
1 myappend([],X,X).
2 myappend([H|T],X,[H|Z]) :- myappend(T,X,Z).
3 ?- myappend([2,3,4],[5,6,7],X).
4 X = [2, 3, 4, 5, 6, 7] ;
5 No
6 ?- myappend(X,[2,3],[1,2,3,4]).
7 No
8 ?- myappend(X,[2,3],[1,2,3]).
9 X = [1] ;
10 No
```

Accumulator recursion works in Prolog, too!

```
1 myreverse(X,Y) :- myrevaux(X,Y,[ ]).
2 myrevaux([ ],Y,Y).
3 myrevaux([HX|TX],Y,Z) :- myrevaux(TX,Y,[HX|Z]).
4 ?- myreverse([2,3,4],Y).
5 Y = [4, 3, 2]
```

```
myreverse([2,3,4],Y) → myrevaux([2,3,4],Y,[ ]) →
myrevaux([3,4],Y,[2]) → myrevaux([4],Y,[3,2]) →
myrevaux([ ],Y,[4,3,2]) →
myrevaux([ ],[4,3,2],[4,3,2]) →
myreverse([2,3,4],[4,3,2])
```

- Write the Fibonacci predicate. Let  $F_0 = 0$  and  $F_1 = 1$ .
- Make sure you can write it the exponential way.
- Can you write it the linear way?

- The term `socrates` is a pattern. But patterns can have structure....

```
1 pair((X,Y)).
2 key((X,Y),X).
3 value((X,Y),Y).
4 assoc(X,Y,[H|T]) :- key(H,X), value(H,Y);
5                      assoc(X,Y,T).
6 ?- assoc(2,X,[(3,hi),(4,there),(2,guys)]).
7 X = guys
8 ?- assoc(X,there,[(3,hi),(4,there),(2,guys)]).
9 X = 4
```

- Fibonacci predicate: exponential complexity:

```
1 fib(0,0).
2 fib(1,1).
3 fib(N,X) :- N1 is N - 1,
4             fib(N1,X1), N2 is N - 2,
5             fib(N2,X2), X is X1 + X2.
```

- Fibonacci predicate: linear complexity:

```
1 lfibx(0,F1,F2,A) :- A is F2.
2 lfibx(N,F1,F2,A) :- N1 is N - 1, F3 is F1 + F2,
3                     lfibx(N1,F2,F3,A).
4 lfib(N,A) :- lfibx(N,1,0,A).
```